# 5.springboot框架

## 5.1.单体项目的其他问题

### 5.1.1.功能的强耦合

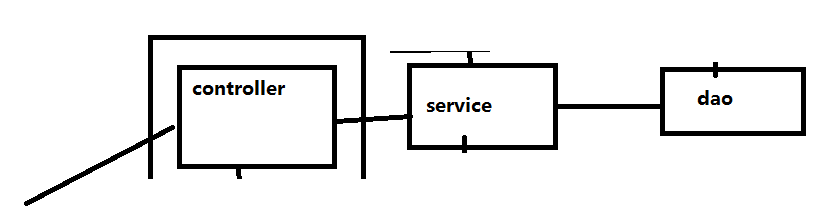
由于所有业务都在一个项目中，随着业务模块的增加，系统会变得越来越复杂，一旦某些功能存在强耦合，会造成开发人员必须精通多个业务模块逻辑，多个领域知识，不利于团队、人员配合，也不利于项目扩展

需要将不同的功能和业务进行拆分，例如order-user项目，可以拆分为order和user两个项目，这样，order项目只关心和订单相关业务，不关心user相关业务，反之亦然，这样的功能拆分称之为纵向拆分

### 5.1.2.系统拆分

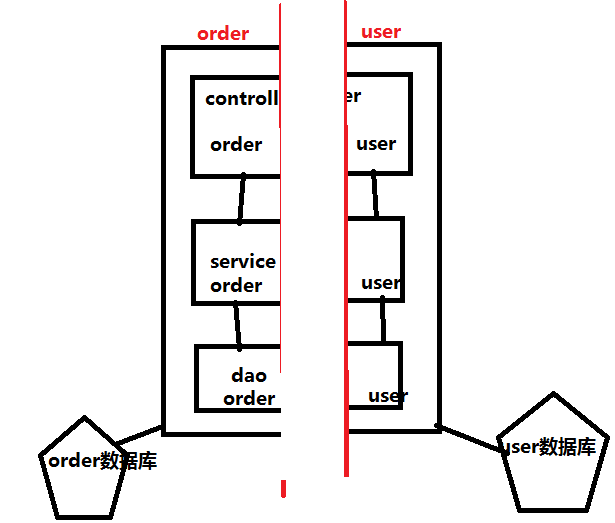
1.横向拆分（解决项目开发的强耦合）

根据项目的三层结构，控制层依赖业务层，依赖持久层，进行分模块的开发，不同的程序员开发不同的模块



2.纵向拆分(解决业务模块功能强耦合)

就是解决业务功能系统不同部分拆分成独立运行的系统项目，相互之间数据调用根据接口文件访问



### 5.1.3.并发集中

如果是单体项目，很有可能某一个功能并发量特别大，导致整系统响应慢，如果纵向拆分，即使某个系统并发量特别大，也不会影响别的系统的正常访问

## 5.2.springboot

### 5.2.1.背景

nginx解决了并发问题，但是没有解决并发集中问题，功能强耦合，需要进行拆分（纵向拆分），这种情况下，一种方便的工具框架就产生了-springboot

### 5.2.2.什么是springboot

一种基于spring框架，完成独立运行，自动配置的一种工具框架。

### 5.2.3.springboot的特点（相对web应用）

**1.独立运行的spring项目**

springboot封装了spring所有的启动过程，可以通过简单的方法实现启动一个具有spring框架功能的进程，如果不做web应用，不需要tomcat这种外部容器来加载spring配置文件，从而生成spring容器的过程，因为spring4.X版本的出现，几乎所有的配置都转换成了注解

**2.springboot内嵌的servlet**

不配置任何tomcat插件，不用将项目发布到任何外部的tomcat容器，直接在依赖总完成了默认的servlet容器整合，一旦启动了spring的独立运行机制，可以加载内嵌的servlet实现web应用的独立运行

**3.springboot提供简化依赖**

根据开发人员依赖的不同，早就封装了简化依赖，实现依赖的传递性，例如：当开发一个web应用是，暂不考虑持久层，只需要依赖一个spring-boot-starter-web，将会给你传递tomcat、spring-beans、jackson、log4g等非常多的jar包

**4.springboot自动配置**

默认自动的扫描范围

启动springboot工程，所有扫描范围内的注解都会加载到内存

根据依赖完成相关自动配置

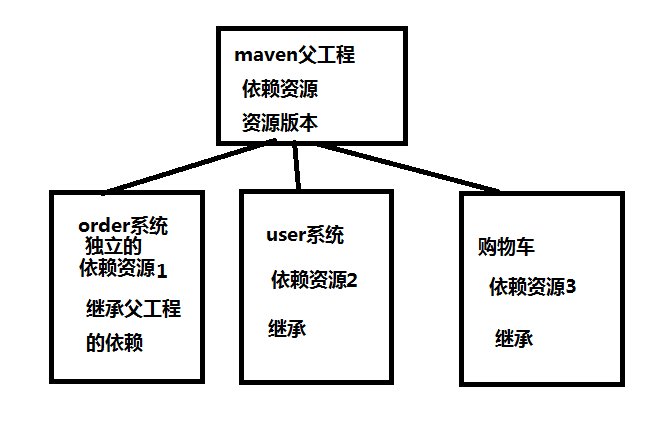
springboot可以根据项目中完成的依赖资源，例如spring-boot-starter-web，自动配置就会判断功能是开发一个web应用，就会完成tomcat的自动加载，默认端口8080，还有其他配置信息等

以上特点导致springboot可以“0”配置启动一个web应用

## 5.3.maven继承

### 5.3.1.继承的意义

当项目进行拆分后，搭建的项目越来越多，由于每个项目都可能依赖了同一个资源，但是版本不同，导致最终整合时，产生冲突，可以通过maven继承解决这个资源管理问题



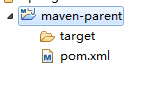
### 5.3.2.搭建父工程

**1.创建父工程**

创建一个quickstart的maven工程（maven-parent）

父工程的意义就是管理依赖资源，所以，除了pom.xml，其他可以删除

packaging必须是pom



<groupId>cn.tedu</groupId>

<artifactId>maven-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>pom</packaging>

**2.创建子工程**

创建一个quickstart的maven工程（maven-child01）

需要在pom文件中添加继承的标签

添加后，子工程的groupId、version都可以移除

<artifactId>maven-child01</artifactId>

<packaging>jar</packaging>

<parent>

<groupId>cn.tedu</groupId>

<artifactId>maven-parent</artifactId>

<version>0.0.1-SNAPSHOT</version>

</parent>

### 5.3.3.子工程可以从父工程继承的资源

groupid:可以继承，也可以覆盖

version：可以继承，也可以覆盖（一般不会覆盖）

properties：可以继承，例如父工程中定义了一些变量和值可以被子工程继承

dependencies：可以继承（强制继承），一旦父工程依赖了一些资源，子工程继承后会进入 子工程可以直接使用

dependencyManagemeng:声明式依赖继承，子工程不会直接继承，需要写依赖指向groupId、 artifactid后才能继承，作用就是统一资源的版本

build：所有的插件也可以继承，例如父工程定义了打源码包的插件

Organization：组织者

## 5.4.手动创建springboot的入门案例

### 5.4.1.案例需求

创建一个项目，发起请求传递参数String name，返回响应体字符串在浏览器显示：“hello XXX springboot”

### 5.4.2.搭建步骤

**1.没有页面，创建quickstart的maven骨架（springboot-test01）**

**2.在pom文件中继承和依赖资源**

继承springboot的parent

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.9.RELEASE</version>

</parent>

引入springboot开发web应用的核心依赖

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

**3.编写项目的启动测试类**

|  |
| --- |
| **package** cn.tedu;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  /\*\*  \* springboot工程的启动入口类，包含main方法，  \* 调用springboot开发的独立运行spring容器的逻辑  \*/  @SpringBootApplication  **public** **class** StarterSpringbootTest01 {  **public** **static** **void** main(String[] args) {  //springboot封装的run方法，可以加载当前运行的环境，  //创建spring框架的底层内容,实现spring容器的独立运行  SpringApplication.*run*(StarterSpringbootTest01.**class**, args);  }  } |

**4.编写请求响应逻辑**

HelloController

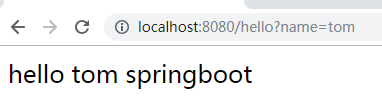
|  |
| --- |
| **package** cn.tedu.controller;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  /\*\*  \* 测试工程的Controller，spring3.X出现了组合注解的技术，  \* 例如@RestController，则当前类中的所有方法无需添加@ResponseBody注解了  \*  \*/  @RestController  **public** **class** HelloController {  @Autowired  **private** HelloService helloService;  @RequestMapping("hello")  **public** String sayHi(String name){  **return** helloService.sayHi(name);  }  } |

HelloService

|  |
| --- |
| **package** cn.tedu.service;  **import** org.springframework.stereotype.Service;  @Service  **public** **class** HelloService {  **public** String sayHi(String name) {  **return** "hello "+name+" springboot";  }  } |

启动类右键->run as->java application即可启动

浏览器访问http://localhost:8080/hello?name=tom



## 5.5.springboot运行原理

### 5.5.1.spring衍生的注解

@ComponentScan

前提是程序能扫描加载到此注解，可以代替spring.xml中的<context:component-scan base-package=”cn.tedu”>标签，@ComponentScan所在的包，就是base-package的值

@Configuration

前提是程序能扫描加载到此注解，将一个spring.xml的配置文件的内容转化成代码内容完成的注解，可以代替一个独立的xml中所需要的<bean>，@Bean注解可以生成一个bean

案例：通过配置类加载对象

配置类

|  |
| --- |
| **package** cn.tedu.config;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** cn.tedu.bean.test.BeanTest01;  /\*\*  \* 相当于一个xml文件-spring.xml  \*/  @Configuration  **public** **class** BeanConfig {  //创建一个BeanTest01的IOC控制反转对象  @Bean//和@Configuration配合使用  //方法的返回值就是需要创建的bean对象  **public** BeanTest01 initBeanTest01(){  //初始化对象，为对象的属性赋值  **return** **new** BeanTest01();  }  } |

需要生成的对象类

|  |
| --- |
| **package** cn.tedu.bean.test;  **public** **class** BeanTest01 {  **public** BeanTest01(){  System.***out***.println("BeanTest01被加载啦。。。");  }  } |

再次启动springboot时，会打印“BeanTest01被加载啦。。。”，说明BeanTest01被创建了

@ConfigurationProperties

前提是程序能扫描加载到此注解，可以通过这个注解读取properties的配置文件中的key=value属性内容，一般配合@Component注解使用

案例：利用此注解实现配置文件中key=value的读取

首先准备一个配置文件在src/main/resources中：application.properties

cn.tedu.name=zhangsan

cn.tedu.className=1906

然后创建一个Bean类

|  |
| --- |
| **package** cn.tedu.bean.test;  **import** org.springframework.boot.context.properties.ConfigurationProperties;  **import** org.springframework.stereotype.Component;  @Component //容器一旦扫描，对象就会创建  @ConfigurationProperties(prefix="cn.tedu")//读取配置文件properties的  **public** **class** ConfigProperties {  **public** ConfigProperties(){  System.***out***.println("ConfigProperties被加载了...");  }  **private** String name;  **private** String className;  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** String getClassName() {  **return** className;  }  **public** **void** setClassName(String className) {  **this**.className = className;  }  } |

最后，可以在Controller中注入此类，然后方法中打印输出属性，发现可以打印出来

@ConditionalOnClass(A.class,B.class)

spring4.X出现的一个重要的自动配置衍生注解，是一个条件注解，判断当前容器中是否有A.class和B.class，如果这两个类同时存在，此类才会被加载，配合@Configuration使用

案例：根据当前环境中存在的不同条件，创建不同的bean对象

创建一个有条件的配置类

|  |
| --- |
| **package** cn.tedu.config;  **import** org.springframework.boot.autoconfigure.condition.ConditionalOnClass;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.context.annotation.Configuration;  **import** cn.tedu.bean.test.ConditionBeanA;  /\*\*  \* 一旦加入条件注解，当前类是否执行，取决于条件注解中的类是否存在  \* 如果存在，可以执行此类，反之亦然  \*/  @Configuration  @ConditionalOnClass(ConditionalA.**class**)  **public** **class** ConditionalConfig {  @Bean  **public** ConditionBeanA initA(){  **return** **new** ConditionBeanA();  }  } |

创建一个@Bean需要创建的类

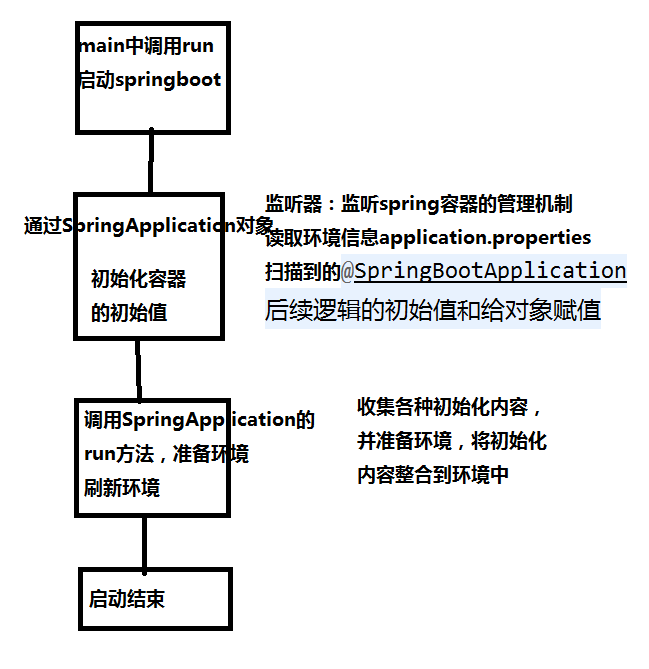
|  |
| --- |
| **package** cn.tedu.bean.test;  **public** **class** ConditionBeanA {  **public** ConditionBeanA(){  System.***out***.println("ConditionBeanA被加载啦....");  }  } |

测试结果：由于有判断注解，而判断注解中的判断类不存在，所以判断注解所在的类就无法执行

如果创建出判断注解中的判断类，则此类会被执行

### 5.5.2.springboot启动顺序

run方法加载顺序：



### 5.5.3.@SpringBootApplication注解

这是一个组合注解包含了以下注解：

**@ComponentScan**

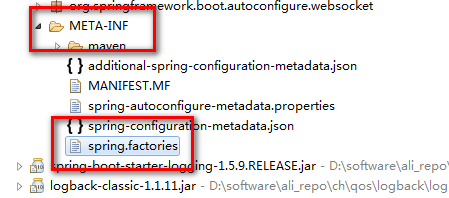
启动类上的核心注解之一，对自定义的代码实现扫描范围的约束（扫描范围就是启动类所在包的同级和子集包）

**@EnableAutoConfiguration**

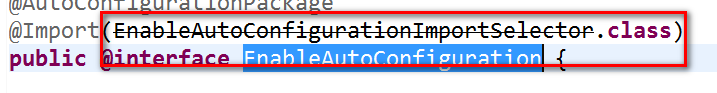
开启自动配置，所有依赖中需要配置的内容，都会在传入的一个\*.autoconfigure.\*的jar包中



在META-INF（元信息）下有一个spring.factories文件，此文件包含了当前依赖的自动配置想要配置的所有配置类信息



@EnableAutoConfiguration会注入一个selector类



进入此类的父类，父类有一个核心方法：selectImports，此方法可以导入所有的配置文件信息

**@SpringBootConfiguration**

是一个标识启动类，也是配置类

作用：读取springboot相关的配置文件，读取的配置文件有：

1.application.properties：全局配置文件

2.banner.txt：启动横幅

3.bootstrap.properties：配置客户端加载的文件，用不到

### 5.5.4.springboot启动原理总结

背景：需要简化配置，spring4.x将所有xml简化到注解（java1.5出现注解技术）

启动顺序：SpringApplication类初始化数据资源，然后调用run方法准备环境和刷新环境

核心配置注解：

@ComponentScan：扫描范围约束、

@EnableAutoConfiguration：自动配置，可以根据依赖内容加载该加载和创建的类

@SpringBootConfiguration：加载配置文件

实现结果：

自动配置

自动扫描代码中所有和spring、springmvc相关的注解

根据依赖完成该配置的内容

昨日回忆：

nginx负载均衡：

轮询

权重

ip\_hash

springboot框架

maven的继承

背景

手动搭建springboot

运行原理：

调用run启动->初始化资源->准备环境->启动完成

@SpringBootApplication：组合注解

## 5.6.springboot配置文件

### 5.6.1.application.properties

springboot的全局配置文件，所有可以使用的key=value值都可以在这个文件中配置，默认的扫描路径是classpath的根目录-maven项目的src/main/resources

例如：可以配置全局参数-tomcat端口、工程访问的默认路径

server.port=80

server.contextPath=/test

在此配置文件中加入以上两行代码，则访问路径变为

<http://localhost/test/hello?name=abc>

还可以配置mybatis、datasource相关配置信息

### 5.6.2.application.yml

和application.properties作用完全一致，唯一的区别就是数据格式不同，yml文件和properties文件不能同时存在，有我无他

yml文件格式：

例如，将： cn.tedu.name=zhangsan

cn.tedu.className=1906

cn.tedu.student.score=99

转化为yml格式：

cn:

[空格]tedu:

[空格][空格]className:[空格]1906

[空格][空格]name:[空格]zhangsan

[空格][空格]student:

[空格][空格][空格]score:[空格]99

eclipse等开发工具对yml格式，高版本的eclipse直接支持，有的版本需要安装yml插件

### 5.6.3.banner.txt

启动横幅，作为企业、公司、项目启动的标识

案例：改变启动标识

在resources下创建banner.txt文本，内容自定义，然后启动springboot查看启动横幅

### 5.6.4.额外配置

springboot虽然准备了丰富的自动配置环境，但是在复杂的开发场景中，不一定能100%满足，可能依然要用到xml进行配，此时，就可以手动导入xml，通过@Configuration和@ImportResource(xml资源位置)

@ImportResource(xml资源位置)注解中，xml资源位置可以有多个，例如：

value={“classpath:spring.xml”,”classpath:abc.xml”}

案例：将一个扫描范围之外的bean对象创建到xml的bean标签

首先，准备一个bean

|  |
| --- |
| **package** com.jt;  **public** **class** People {  **public** People(){  System.***out***.println("people登场了.....");  }  } |

准备一个xml（复制之前的一个spring.xml，只留头）

<bean class=*"com.jt.People"*></bean>

最后在配置类上加入注解

例如，在BeanConfig类上加入注解：

@Configuration

@ImportResource("classpath:spring.xml")

启动springboot，发现可以加载people类

## 5.7.mybatis持久层整合

需求：查询student

### 5.7.1.springboot实现自动配置-依赖

引入依赖spring-jdbc、mybatis、mysql

|  |
| --- |
| <!-- jdbc -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  </dependency>  <!-- mysql -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <!-- mybatis -->  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>1.3.0</version>  </dependency> |

### 5.7.2.配置文件：application.properties

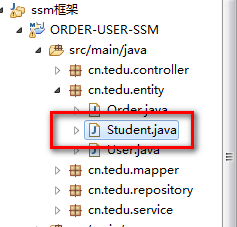
|  |
| --- |
| #datasource  spring.datasource.driverClassName=com.mysql.jdbc.Driver  spring.datasource.url=jdbc:mysql:///test  spring.datasource.username=root  spring.datasource.password=root  #sqlsession  mybatis.typeAliasesPackage=cn.tedu.entity  mybatis.mapperLocations=classpath:mapper/\*.xml  mybatis.configuration.mapUnderscoreToComelCase=true |

### 5.7.3.student数据库查询功能实现

环境已经准备完毕

代码和映射文件的开发（拷贝）

cn.tedu.entity(拷贝到springboot-test01项目的cn.tedu.entity包下)



cn.tedu.mapper(接口类)



mapper映射文件（连同目录）拷贝



StudentController(包含StudentService)

|  |
| --- |
| **package** cn.tedu.controller;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  **import** cn.tedu.entity.Student;  **import** cn.tedu.mapper.StudentMapper;  @RestController  **public** **class** StudentController {  @Autowired  **private** StudentMapper studentMapper;  @RequestMapping("queryStudent")  **public** Student queryStudentById(String studentId){  **return** studentMapper.queryStudentById(studentId);  }  } |

启动类启动，在启动类中添加@MapperScan(“cn.tedu.mapper”)

|  |
| --- |
| **package** cn.tedu;  **import** org.mybatis.spring.annotation.MapperScan;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  @MapperScan("cn.tedu.mapper")  **public** **class** StarterSpringbootTest01 {  **public** **static** **void** main(String[] args) {  //springboot封装的run方法，可以加载当前运行的环境，  //创建spring框架的底层内容,实现spring容器的独立运行  SpringApplication.*run*(StarterSpringbootTest01.**class**, args);  }  } |

## 5.8.springboot解析jsp

### 5.8.1.jsp和springboot的关系

jsp本质就是一个servlet，jsp最终需要翻译成servlet，但是springboot内嵌的tomcat无法直接解析jsp文件，需要添加jsp的依赖资源来解决

### 5.8.2.与jsp的整合步骤

**（1）添加jsp的依赖**

创建一个maven-webapp工程（springboot-jsp），pom中添加父工程和相关依赖

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.9.RELEASE</version>  </parent>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!-- 加入解析jsp的依赖 -->  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>jstl</artifactId>  </dependency>  <dependency>  <groupId>org.apache.tomcat.embed</groupId>  <artifactId>tomcat-embed-jasper</artifactId>  <version>8.5.11</version>  </dependency>  </dependencies> |

**（2）启动文件**

|  |
| --- |
| **package** cn.tedu;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  **public** **class** StarterJsp {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(StarterJsp.**class**, args);  }  } |

**（3）准备Controller**

|  |
| --- |
| **package** cn.tedu.controller;  **import** org.springframework.stereotype.Controller;  **import** org.springframework.web.bind.annotation.RequestMapping;  @Controller  **public** **class** IndexController {  @RequestMapping("/")  **public** String toIntex(){  **return** "index";  }  } |

**（4）配置前后缀**

在application.properties中配置：

spring.mvc.view.prefix=/WEB-INF/

spring.mvc.view.suffix=.jsp

最后，启动项目，访问http://localhost:8080/即可访问到web-inf下的index.jsp页面

## 5.9.springboot的发布运行

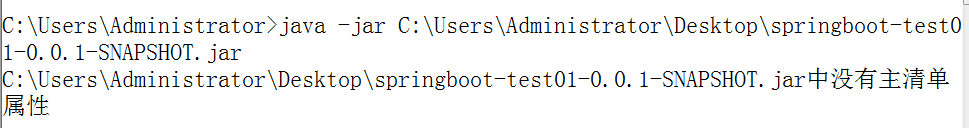
### 5.9.1.打包运行

springboot独立运行的spring容器，可以在jdk环境中，调用java命令运行

打包：项目右键->run as->mvn package(maven install)打包文件在target目录中

### 5.9.2.main方法所在类

如果直接运行jar包，则无法运行：



原因是运行时无法找到main方法，所以需要添加main方法的指定插件，springboot为maven提供了打包springboot的所有插件简化资源

**添加springboot为maven提供的插件资源**

|  |
| --- |
| <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build> |

最后，在cmd窗口中通过java -jar \*.jar命令执行

## 5.10.将ssm框架的ORDER-USER-SSM项目拆分

**纵向拆分**

根据系统的功能进行划分，order（订单支付）功能、user（用户）功能

### 5.10.1.搭建两个maven工程-order

两个maven的quickstart工程-springboot-order、springboot-user

springboot-order工程：

**pom.xml：**

添加parent

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>1.5.9.RELEASE</version>  </parent> |

<properties>标签中添加java1.8版本

|  |
| --- |
| <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <java.version>1.8</java.version>  </properties> |

依赖导入-web应用、jdbc、mybatis、mysql

|  |
| --- |
| <!-- web应用 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!-- jdbc -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-jdbc</artifactId>  </dependency>  <!-- mysql -->  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <!-- mybatis -->  <dependency>  <groupId>org.mybatis.spring.boot</groupId>  <artifactId>mybatis-spring-boot-starter</artifactId>  <version>1.3.0</version>  </dependency> |

配置文件application.properties

|  |
| --- |
| server.port=8092  spring.datasource.driverClassName=com.mysql.jdbc.Driver  spring.datasource.url=jdbc:mysql:///test  spring.datasource.username=root  spring.datasource.password=root  mybatis.typeAliasesPackage=cn.tedu.entity  mybatis.mapperLocations=classpath:mapper/\*.xml  mybatis.configuration.mapUnderscoreToCamelCase=true  mybatis.configuration.cacheEnabled=false |

启动类:

注意，因为整合了mybatis，所以需要多添加@MapperScan注解

|  |
| --- |
| **package** cn.tedu;  **import** org.mybatis.spring.annotation.MapperScan;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  @MapperScan("cn.tedu.mapper")  **public** **class** StarterOrder {  **public** **static** **void** main(String[] args) {  SpringApplication.run(StarterOrder.**class**, args);  }  } |

### 5.10.2.根据接口文件编写系统功能

|  |  |
| --- | --- |
| nginx请求地址 | localhost/order/pay?orderId=a |
| 后台接收地址 | /order/pay |
| 请求方式 | get |
| 请求参数 | String orderId 表示订单id |
| 返回数据 | “1”表示成功，其他表示失败 |

开发顺序：

Order:

|  |
| --- |
| **package** cn.tedu.entity;  **public** **class** Order {  **private** String orderId;  **private** Integer orderMoney;  **private** String userId;  get&set  } |

OrderController

|  |
| --- |
| **package** cn.tedu.controller;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  @RestController  **public** **class** OrderController {  @Autowired  **private** OrderService orderService;  @RequestMapping("/order/pay")  **public** Integer orderPay(String orderId){  **try** {  orderService.orderPay(orderId);  **return** 1;  } **catch** (Exception e) {  e.printStackTrace();  **return** 0;  }  }  } |

OrderService

打印支付的money，将支付的money交给user系统（也要将userId传递过去）

第一版：无数据传递

|  |
| --- |
| **package** cn.tedu.service;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.stereotype.Service;  @Service  **public** **class** OrderService {  @Autowired  **private** OrderMapper orderMapper;  **public** **void** orderPay(String orderId) {  Order order = orderMapper.queryOrder(orderId);  System.***out***.println("订单已经支付"+order.getOrderMoney()+"钱");  //**TODO** 将order对象中的orderMoney传给user系统  }  } |

OrderMapper接口

|  |
| --- |
| **package** cn.tedu.mapper;  **import** cn.tedu.entity.Order;  **public** **interface** OrderMapper {  Order queryOrder(String orderId);  } |

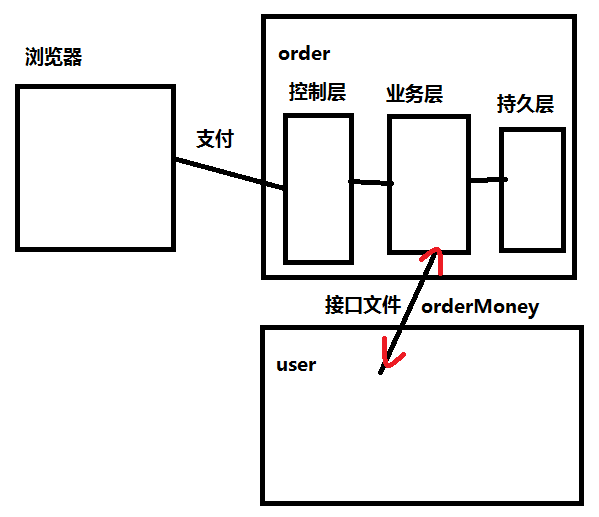
OrderMapper.xml映射文件（可以拷贝之前order-user系统的映射文件）

|  |
| --- |
| <mapper namespace=*"cn.tedu.mapper.OrderMapper"*>  <select id=*"queryOrder"* resultType=*"Order"* parameterType=*"String"*>  select \* from t\_order where order\_id=#{orderId}  </select>  </mapper> |

拆分系统之间的数据交互，由于不同的团队开发不同的系统，为了解决沟通问题，这里也需要定义接口文件

|  |  |
| --- | --- |
| 请求地址: | localhost:8091/user/updatePoint?orderMoney=5000 |
| 请求方式 | get |
| 请求参数 | Integer orderMoney表示支付金额 |
| 返回数据 | 成功1，其他失败，接收数据后进行后续操作 |

系统之间数据交互



需要在service层发起一个http请求，可以通过RestTemplate对象发起

该对象是springmvc底层实现的一个支持http协议的，封装了多个http协议方法的对象

案例：此对象访问京东

RestTemplate测试：

|  |
| --- |
| @Test  **public** **void** test01(){  RestTemplate rt = **new** RestTemplate();  String bodystr = rt.getForObject  ("https://www.jd.com/", String.**class**);  System.***out***.println(bodystr);  } |

**RestTemplate的api：**

getForObject(url, responseType)

此方法是发起一个get请求

参数一：请求路径，参数二：返回值类型，例如返回字符串，则String.class

postForObject(url, request, responseType)

此方法是发起一个post请求

参数一：请求路径，参数三：返回值类型，

参数二：request类型Object

普通类型：比如请求参数是Student对象，这里可以直接写Student对象，如果是字符串，直接写字符串即可，

HttpEntity：表示一个可以使用restTemplate发送的整个request对象，除了可以封装数据，还可以封装请求头等信息

第二版OrderService

|  |
| --- |
| **package** cn.tedu.service;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.stereotype.Service;  **import** org.springframework.web.client.RestTemplate;  **import** cn.tedu.entity.Order;  **import** cn.tedu.mapper.OrderMapper;  @Service  **public** **class** OrderService {  @Autowired  **private** OrderMapper orderMapper;  **public** **void** orderPay(String orderId) {  Order order = orderMapper.queryOrder(orderId);  System.***out***.println("订单已经支付"+order.getOrderMoney()+"钱");  //**TODO** 将order对象中的orderMoney传给user系统  RestTemplate rt = **new** RestTemplate();  Integer success = rt.getForObject(  "http://localhost:8091/user/updatePoint?orderMoney="+  order.getOrderMoney(), Integer.**class**);  }  } |

### 5.10.3.用户系统搭建-springboot-user

1.quickstart的maven项目

2.pom文件（拷贝order系统的pom内容）

3.application.properties（拷贝order的）

注意修改端口为8091

4.启动类StarterUser

|  |
| --- |
| **package** cn.tedu;  **import** org.mybatis.spring.annotation.MapperScan;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  @SpringBootApplication  @MapperScan("cn.tedu.mapper")  **public** **class** StarterUser {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(StarterUser.**class**, args);  }  } |

5.功能编写

准备持久层内容拷贝ORDER-USER-SSM工程的entity、mapper、mapper.xml

根据接口文件，编写user的接口文件

查询积分的接口文件：

|  |  |
| --- | --- |
| 请求地址 | localhost/user/query/point?userId=1 |
| 后台接收地址 | /user/query/point |
| 请求方式 | get |
| 请求参数 | String userId |
| 返回数据 | 对象的json字符串，至少包含一个points属性 |
| 备注 | select \* from t\_user where user\_id=#{userId} |

UserController

|  |
| --- |
| **package** cn.tedu.controller;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.web.bind.annotation.RequestMapping;  **import** org.springframework.web.bind.annotation.RestController;  **import** cn.tedu.entity.User;  @RestController  **public** **class** UserController {  @Autowired  **private** UserService userService;  @RequestMapping("/user/query/point")  **public** User queryUserById(String userId){  **return** userService.queryUserById(userId);  }  } |

UserService

|  |
| --- |
| **package** cn.tedu.service;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.stereotype.Service;  **import** cn.tedu.entity.User;  **import** cn.tedu.mapper.UserMapper;  @Service  **public** **class** UserService {  @Autowired  **private** UserMapper userMapper;  **public** User queryUserById(String userId) {  **return** userMapper.queryUser(userId);  }  } |

### 5.10.4.与页面整合

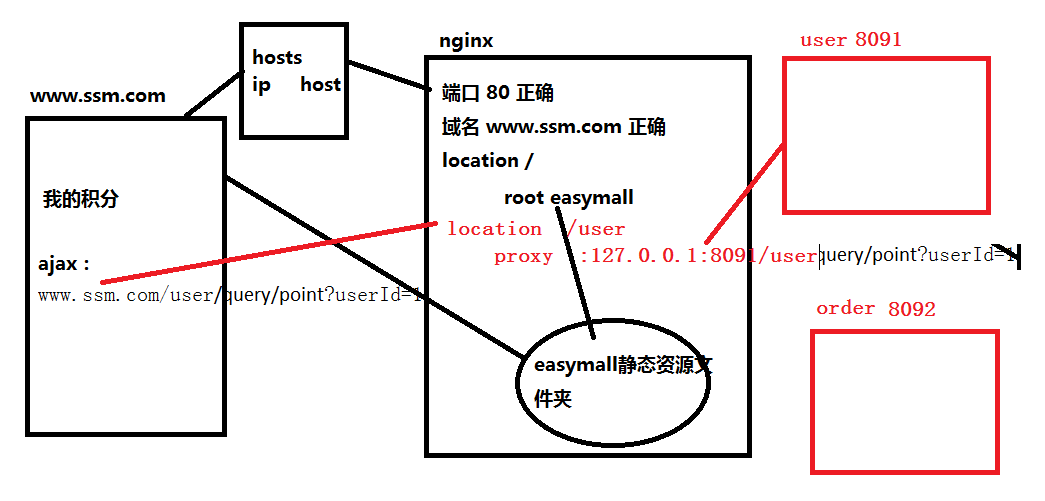
nginx保存了静态页面

需要在hosts中绑定127.0.0.1 [www.ssm.com](http://www.ssm.com)

转发逻辑提前已经配置好，无需改动(nginx.conf)

|  |
| --- |
| server {  listen 80;  server\_name www.ssm.com;  location /user {  proxy\_pass http://127.0.0.1:8091/user;  add\_header 'Access-Control-Allow-Origin' '\*';  add\_header 'Access-Control-Allow-Credentials' 'true';  }  location /order {  proxy\_pass http://127.0.0.1:8092/order;  add\_header 'Access-Control-Allow-Origin' '\*';  add\_header 'Access-Control-Allow-Credentials' 'true';  }  location /{  root easymall;  index index.html;  }  } |

访问逻辑



根据修改积分的接口文件，编写user积分逻辑

UserController中添加一个方法

|  |
| --- |
| @RequestMapping("user/updatePoint")  **public** Integer updatePoint(@RequestParam(value="orderMoney")Integer money){  **try** {  userService.updatePoint(money);  **return** 1;  } **catch** (Exception e) {  **return** 0;  }  } |

UserService添加一个方法

|  |
| --- |
| **public** **void** updatePoint(Integer money) {  userMapper.updateUserPoint(money);  } |

### 5.10.5.当前代码优化

**优化一：对象交给框架**

不能在代码中出现new关键字，由于高并发，可能导致new对象过多，从而内存溢出，需要将RestTemplate对象交由spring框架维护

在一个配置类中完成RestTemplate的创建（可以直接在启动类中）

添加如下代码：

|  |
| --- |
| **package** cn.tedu;  **import** org.mybatis.spring.annotation.MapperScan;  **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  **import** org.springframework.context.annotation.Bean;  **import** org.springframework.web.client.RestTemplate;  @SpringBootApplication  @MapperScan("cn.tedu.mapper")  **public** **class** StarterOrder {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(StarterOrder.**class**, args);  }  @Bean  **public** RestTemplate initRestTemplate(){  **return** **new** RestTemplate();  }  } |

在UserService中，删除之前的创建代码，改为注入

@Autowired

**private** RestTemplate rt;

**优化二：系统访问耦合优化**

orderService访问user系统时，路径写死了，改为：

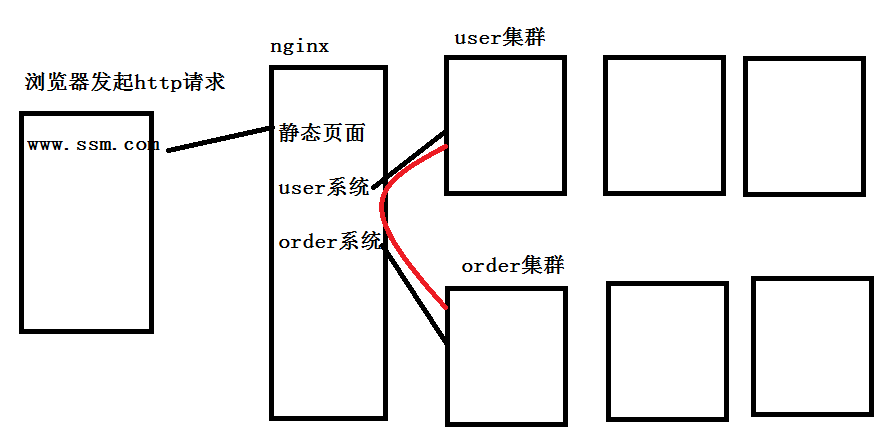
Integer success = rt.getForObject(

"http://www.ssm.com/user/updatePoint?orderMoney="+

order.getOrderMoney(), Integer.**class**);

访问交给nginx，就不是直接访问具体的服务器了

### 5.10.6.当前拆分的访问逻辑



## 5.11.分布式系统服务调用的问题总结

**对集群的管理问题**

对于集群中，某个系统资源空闲的判断，需要引入监听器

发起http请求后，由于调用其他系统时，频繁出现错误，导致压力堆积

**强耦合问题**

nginx配置内容，需要根据服务器的变动而发生修改，配置与服务器绑定在了一起，产生了强耦合，不利于维护